



Using libpolymake_julia

polymake-workshop 2022

Benjamin Lorenz

Technische Universität Berlin
lorenz@math.tu-berlin.de

21.01.2022

polymake structure

The usual polymake consists of

- ◆ a (small) perl script `polymake`
- ◆ which loads several perl modules for
 - ◆ managing (`polymake-`)objects, properties, rules
 - ◆ the shell, the scheduler
 - ◆ and some more technical stuff
- ◆ lots of `.rules` files (parsed by the perl modules)
- ◆ several shared libraries (`.so`) for
 - ◆ hacking into the perl interpreter
 - ◆ the C++-clients of each application (including common)

libpolymake consists of

- ◆ one shared library `libpolymake.so` which is linked against
- ◆ `libperl.so` to load all perl modules and then the rule-base
- ◆ which again load the other shared libraries of all applications

Using libpolymake in five steps

- ◆ `#include <polymake/Main.h>`
- ◆ initialize polymake by creating an instance of `polymake::Main`
- ◆ set an application
- ◆ write code similiar to any polymake C++ client
- ◆ compile and link it using flags from `polymake-config`

one-slide-example

```
#include "polymake/Main.h"
#include "polymake/Rational.h"
using namespace polymake;
int main()
{
    try {
        Main pm("none");
        pm.set_application("polytope");
        BigObject c = call_function("cube", 3);
        BigObject bip = call_function("bipyramid", c, 3, -4);
        const Int nfacets = bip.give("N_FACETS");
        cout << "nfacets: " << nf << endl;
    }
    catch (const std::exception& ex) {
        cerr << "ERROR: " << ex.what() << endl; return 1;
    }
    return 0;
}

$($PMCONF --cc) $($PMCONF --cflags) $($PMCONF --includes) \
-o example example.cc $($PMCONF --ldflags) $($PMCONF --libs)
```

Interface between julia and polymake

Components:

- ◆ `julia`
- ◆ `Polymake.jl julia`
- ◆ `CxxWrap.jl julia`
- ◆ `libcxxwrap_julia C++`
- ◆ `libpolymake_julia C++`
- ◆ `libpolymake C++`
- ◆ `polymake perl & C++`

polymake in julia

```
julia> using Polymake
polymake version 4.6
Copyright (c) 1997-2021
Ewgenij Gawrilow, Michael Joswig, and the polymake team
Technische Universität Berlin, Germany
https://polymake.org
```

This is free software licensed under GPL; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

```
julia> c = polytope.cube(3);
```

```
julia> c.F_VECTOR
pm::Vector{pm::Integer}
8 12 6
```

libpolymake_julia core

```
#include <*>
JLCXX_MODULE define_module_polymake(jlcxx::Module& jlpolymake)
{
    add_bigobject(jlpolymake);
    add_matrix(jlpolymake);
    add_direct_calls(jlpolymake);
    // ...

    jlpolymake.method("initialize_polymake",
                      &initialize_polymake);
    jlpolymake.method("application", [] (const std::string x) {
        data.main_polymake_session->set_application(x);
    });
}
```

polymake types in libpolymake_julia

```
julia> a = Polymake.Array{Int}(3);
```

```
julia> length(a)
```

```
3
```

```
julia> a[1] = 3
```

```
3
```

```
julia> a[2]
```

```
0
```

```
julia> a
```

```
pm::Array{long}
```

```
3 0 0
```

```
julia> a[4]
```

```
ERROR: BoundsError: attempt to access 3-element Polymake.ArrayAllocated{Int
```

```
Stacktrace:
```

```
[1] throw_boundserror(A::Polymake.ArrayAllocated{Int64}, I::Tuple{Int64})
```

```
@ Base ./abstractarray.jl:691
```

```
[2] checkbounds
```

```
@ ./abstractarray.jl:656 [inlined]
```


polymake types in libpolymake_julia

```
void add_array(jlcxx::Module& jlpolymake)
{
    auto type = jlpolymake.add_type<
        jlcxx::Parametric<jlcxx::TypeVar<1>>>>(
            "Array", jlcxx::julia_type("AbstractVector", "Base"));
    type.apply<pm::Array<pm::Int>>([](auto wrapped) {
        typedef typename decltype(wrapped)::type WrappedT;
        typedef typename WrappedT::value_type elemType;
        wrapped.template constructor<int64_t>();
        wrapped.method("length", &WrappedT::size);
        wrapped.method("_getindex", [](const WrappedT& A, int64_t n){
            return elemType(A[static_cast<pm::Int>(n) - 1]);
        });
        wrapped.method("take", [](pm::perl::BigObject p,
            const std::string& s, WrappedT& A) { p.take(s) << A; });
    });
    jlpolymake.method("to_array_int",
        [](const pm::perl::PropertyValue& pv) {
            return to_SmallObject<pm::Array<pm::Int>>(pv);
        });
}
```

polymake functions in libpolymake_julia

How to call polymake functions?

```
julia> c = polytope.cube(3);  
julia> c = Polymake.convert_from_property_value(  
    Polymake.internal_call_function(  
        "polytope::cube",  
        Polymake.CxxWrap.StdVector{  
            Polymake.CxxWrap.StdString}  
        (String[]),  
        Any[3]  
    ))
```

When building `polymake_jll` we generate json files describing all polymake `user_functions` from which we generate these function wrappers during precompilation.

one-slide-example version 2

```
#include "polymake/Main.h"
#include "polymake/Rational.h"
using namespace polymake;
int main()
{
    try {
        Main pm("none");
        pm.set_application("polytope");
        BigObject c = call_function("cube", 3);
        auto bip_call = prepare_call_function("bipyramid");
        bip_call << c; bip_call << 3; bip_call << Rational(-4);
        BigObject bip = bip_call();
        auto nfacets_pv = bip.give("N_FACETS");
        const Int nf = static_cast<Int>(nfacets_pv);
        cout << "nfacets: " << nf << endl;
    }
    catch (const std::exception& ex) {
        cerr << "ERROR: " << ex.what() << endl; return 1;
    }
    return 0;
}
```

polymake functions in libpolymake_julia

```
template<typename return_type=typename context::scalar>
auto polymake_call_function(
    const std::string&                function_name,
    const std::vector<std::string>&    template_vector,
    const jlcxx::ArrayRef<jl_value_t*, 1> arguments)
-> return_type
{
    auto function = polymake::prepare_call_function(
        function_name, template_vector);
    for (auto arg : arguments)
        call_function_feed_argument(function, arg);
    return static_cast<return_type>(function());
}

void add_caller(jlcxx::Module& jlpolymake)
{
    jlpolymake.method("internal_call_function",
        &polymake_call_function<>);
    // ...
}
```

Infrastructure around julia

- ◆ `julia`

- ◆ `Pkg.jl`

To install `Polymake.jl` and dependencies,
either directly from github or released versions via the julia registry.

- ◆ Binary dependencies cross-compiled with `BinaryBuilder.jl`,
recipes in `Yggdrasil` (<https://github.com/JuliaPackaging/Yggdrasil>)
`polymake_jll`, `libpolymake_julia_jll`, `libcxxwrap_jll`,
`GMP_jll`, `FLINT_jll`, ...